(1)

USER'S MANUAL(Part II)

A MICROCODE COMPILER THAT RUNS ON THE IBM AT AND SUPPORTS CASCADABLE MICROCOMPUTERS

by

Thomas H. Weight,Ph.D.

Period Covered: 23 Sept 87 to 22 Sept 89

Contract DAAD07-87-C-0119
for
White Sands Missile Range, New Mexico 88002

11 Nov 89

PENGUIN SOFTWARE,Inc.
7005 E. Spring St.
Long Beach, Calif. 90808

[illegible]
Thomas H. Weight,Ph.D.
Principal Investigator
PENGUIN SOFTWARE,Inc.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS ----
2a. SECURITY CLASSIFICATION AUTHORITY ----	3. DISTRIBUTION/AVAILABILITY OF REPORT pproved for Public Release; istribution is unlimited. PART II
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE ----	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ----	5. MONITORING ORGANIZATION REPORT NUMBER(S) ----
6a. NAME OF PERFORMING ORGANIZATION PENGUIN SOFTWARE, Inc. 6b. OFFICE SYMBOL (If applicable) ----	7a. NAME OF MONITORING ORGANIZATION U.S. ARMY White Sands Missile Range
6c. ADDRESS (City, State, and ZIP Code) 7005 E. Spring St. Long Beach, CA 90808	7b. ADDRESS (City, State, and ZIP Code) COMMANDING OFFICER, STEWS-ID-T U.S. ARMY White Sands Missile Range New Mexico 88002-5143
8a. NAME OF FUNDING/SPONSORING ORGANIZATION ---- 8b. OFFICE SYMBOL (If applicable) ----	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAD07-87-C-0119
8c. ADDRESS (City, State, and ZIP Code) ----	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
665502	1P65502M40	----	----

11. TITLE (Include Security Classification)
A Microcde Compiler that runs on the IBM AT

12. PERSONAL AUTHOR(S)
Thomas H. Weight, PH.D.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Technical	FROM 23Sept87 TO 22Sept89	1989,Nov,11	

16. SUPPLEMENTARY NOTATION
----

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
----	----	----
----	----	----

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
High-level language, microprogramming, automated microcode generation, miercompiler. JES) [illegible]

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

PENGUIN SOFTWARE, Inc. has developed a retargetable microcode compiler. Our approach does not have a fixed machine independent language, but allows the user to develop a language specific to each particular target machine. PENGUIN SOFTWARE"s Microcode Compiler starts out with an underlying meta-assembler and builds up a higher level language capability around it. This capability allows the user to incorporate knowledge of target machine design into the language definition, and thus avoid the necessity for resource allocation and code compaction in the application program. This approach results in a microcode development tool which is a very low risk, very fast, and is capable of supporting virtually any digital hardware architecture.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL FOO LAM	22b. TELEPHONE (Include Area Code) (505)678-3010 22c. OFFICE SYMBOL STEWS-ID-T

DD Form 1473, JUN 86 Previous editions are obsolete. 258

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

intentionally left blank

TABLE OF CONTENTS

DICLAIMER OF WARRANTIES
AND LIMITATION OF LIABILITIES

The staff of PENGUIN SOFTWARE, Inc. has taken due care in preparing this manual and the demonstration program; however this is a preliminary release of both our microcode compiler and its documentation. In no event shall PENGUIN SOFTWARE, Inc. be liable for incidental or consequential damages in connection with or arising from the use of the software, the corresponding manuals, or any related materials.

intentionally left blank

CHAPTER 1

GENERAL INFORMATION

1.1 INTRODUCTION

This user's manual is provided for the PENGUIN SOFTWARE,Inc. microcode compiler hereafter referred to as PSI. This compiler works in conjunction with the HILEVEL TECHNOLOGY Assembly Language Environment Program (HALE). It is assumed that HALE is available to and familiar to the reader.

This software package requires an IBM PC or compatible with a minimum memory size of 640k bytes. The program is supplied in the form of IBM PC compatible diskettes.

1.2 PURPOSE OF PSI

This compiler provides a high-level language capability for the generation of microcode. Conventional compilers will provide a particular language for a particular machine. The PSI compiler differs in two important respects. First, this compiler can be easily retargetted to any machine supported by HALE. Since HALE is a state-of-the-art meta-assembler, this allows PSI to support a wide range of computer architectures. Second, PSI is a meta-compiler. This means that PSI can support several different languages. Microcode compilers which support just one language invariably produce inefficient microcode which has to be compacted and optimized. PSI allows the user to desigh a language optimized to a particular computer and to a particular application. The result is that PSI automatically allocates resources and produces optimized eficicent microcode.

1.3 ORGANIZATION OF PSI

PSI is divided into the following two segments:

1. SYNTAX DEFINITION PHASE. This phase defines the syntax and semantics of the high-level language. This is analogous to the definition phase of the HALE meta-assembler.

2. MICROCODE COMPILER. This phase compiles an application program written in a high-level language into HALE assembly language. The high-level language supported by this compiler is determined by the language defined in the SYNTAX PHASE.

1.4 ORGANIZATION OF THE MANUAL

This manual provides information needed to write a high-level language program and processes it through the PSI compilers. It is assumed that the reader is familiar with the HALE meta-assembler and with microprogramming in general.

Chapter 2 provides an overview of how the PSI microcompiler system works and interfaces with HALE.

Chapters 3 and 4 provide descriptions of the basic elements of the syntax compiler and microcompiler source programs.

Chapters 5 and 6 provide descriptions of the structures of the syntax compiler and microcompiler source programs.

Chapters 7 and 8 provide information on how to run the syntax compiler and microcompiler.

CHAPTER 2

A SIMPLIFIED FIRST LOOK AT COMPILER USAGE

This chapter is included primarily for those who are unfamiliar with meta-compilers. Typically, compilers require a user to become familiar with the language supported and how to operate the compiler. In order to use this system to maximum benefit, it is also necessary to have a certain familiarity with the internal workings of a compiler.

2.1 INTRODUCTION

The purpose of PSI is to provide the user a facility for designing a language tailored for the development of microcode. This normally will involve two steps.

First, a language will be designed which is dependent on the application and the underlying machine organization. Specifying a language requires the specification of the syntax and semantics of the language. The syntax determines which "sentences" and "clauses" are legitimate in the language, while the semantics determines the actions to be taken by the compiler when a sentence is analyzed and recognized (parsed) by the compiler.

The second step involves compiling an application program, written in a language specified in step one, and producing an object file. In the case of PSI, the object file produced will consist of a series of HALE assembler source statements. These statements will be formatted so that they are compatible with the HALE microcode development system. This assembler source file can be modified or processed directly through the HALE meta-assembler to produce the required microcode.

2.2 MICROCODE COMPILER

The statements in an application language consist of either pseudo operations, sentences and comments. The pseudo operations are described in chapter 4. They follow HALE conventions as closely as possible.

The sentences have the following format:

```
<label>: <clause> & <clause> & ... & <clause>; <comment>
```

A sentence can consist of one or more clauses preceeded by an optional label and followed by an optional comment. The semicolon is in fact the only required part of a sentence. A sentence consisting of zero clauses will be considered a comment which may be labeled. Several examples can be found in Appendices A and B.

The clauses are parsed by the compiler to produce the object file. The clauses are defined in the syntax definition phase.

2.3 SYNTAX DEFINITION

The clauses are defined in the syntax phase program by using the CLAUSE, the ARRAY, and the LITERAL peusdo operations. The CLAUSE statement has the following format:

```
CLAUSE {entity entity ... entity}{semantic actions}
```

where the entities are ARRAY's, LITERAL's or literal strings. The format of the literal string is the most basic - just a quoted string.

An example of a CLAUSE definiton consisting of just a literal string might be:

```
     CLAUSE { "NOP" }{semantic action}
```

Thus if the following sentence were encountered:

```
  noontime:    NOP;   take a lunch break
```

then the semantic actions specified according to the rules in the next section would be performed.

The ARRAY is used when anyone of several strings can be used in a CLAUSE. Rather than having to specify a new CLAUSE statement for each string, we have the option a simply specifying the ARRAY name in the CLAUSE definition instead. The ARRAY statement has the following format:

```
ARRAY <ARRAY NAME> = {
             <string>{semantic action}; <comment>
             <string>{semantic action}; <comment>
                          .
                          .
                          .
```

```
                <string>{semantic action}: <comment>
}
```

..s an example of the use of the ARRAY in a CLAUSE definiton, we have the following:

```
     CLAUSE { REGD "=" REGS }

     ARRAY REGD = {
                 R0{semantic action}
                 R1{semantic action}
     }
     ARRAY REGS = {
                 R2{semantic action}
                 R3{semantic action}
     }
```

As a result of this CLAUSE definition, the following sentence would be recognized:

```
                 R1 = R2 ; copy r2 into r1
```

and the associated semantic actions would be performed as required.

In the case where we do not have orthogonality between operands, we can use a dimensioned ARRAY. For example, if the following combinations were legitimate:

```
             R0 = R2;
             R1 = R2;
             R0 = R3;
```

but R1 = R3 were to be excluded, then the preceeding approach would not be acceptable. The dimensioned ARRAY would solve this problem as follows:

```
     ARRAY REGS(1,2) = {
             R0,R2{semantic action}
             R1,R2{semantic action}
             R0,R3{semantic action}
     }
     CLAUSE  {REGS(1) "=" REGS(2)}
```

The dimensioned ARRAY's have the property that if more than one column from a dimension ARRAY appears in a CLAUSE, then all of the strings must come from the same row. The result in this case is that only the required sentences are considered to be legal.

The final syntax entity is the LITERAL. The LITERAL allows the programmer to use literal numbers or labels defineu in the application program. The format of the LITERAL is as follows:

```
        LITERAL(#) = {semantic action}; <commoent>
```

The LITERAL might be used in the following situation:

```
        CLAUSE { REGD "=" LITERAL(1)}
        LITERAL(1) = {semantic action}
```

Now the following sentence could be parsed by the microcode compiler:

```
               R1 = 9;
```

2.4 SEMANTICS DEFINITION

The fundamental semantic action is the HALE statement. Any DEF'ed command can be used in the semantics section of the syntax definition. There are a few restrictions on the HALE Def'ed statements that can be used as semantic actions. The first restriction is that the DEF and SUB statements can not contain any so colled "don't care fields". These can be replaced with variable fields with don't care default values. The other know restriction is that all commas must appear in the instruction/data source statements. The HALE meta-assembler will handle missing commas properly, but the compiler will flag and error if there are either too many or too few commas in a Def'ed statement.

In addition to these DEF'ed statements, we have the internal compiler actions. To understand these actions it is necessary to understand a little of the internal structure of the compiler. The compiler has several resources which can be controled in order to implement various high-level language constructs. These resources are the stack, the active label register, and the internal registers. The semantic actions which control these resources are as follows:

a) LBL appearing in a DEF'ed statement will cause the active label register to be inserted into the corresponding field value.

b) PSH will cause the active label register to be pushed onto the internal stack.

c) PPS will cause the active label register to be poped off the internal stack.

d) MAK will cause a label to be created and stored in the active label register.

e) PLB will cause the label stored in the active label register to be assigned to the corresponding sentence.

f) LBLn (where n = 1,2,3,4,5) will cause the active label register to be stored in internal register n.

g) SWP will cause the top two labels stored in the internal stack to be swapped.

h) LIT used in a LITERAL pseudo operation will cause the corresponding literal to be stored in the active label register. The corresponding literal in this case is the literal located in the same relative position within the CLAUSE as the LITERAL statement.

i) LITn (where n = 1,2,3,4,5) will cause internal register n to be copied to the active label register.

The following simple example should help clarify the situation. Suppose the following CLAUSE is defined in the syntax definition program:

```
CLAUSE {"GOSUB" LITERAL(1)}
         {AM2909    ,JSRFNO,}
LITERAL(1) = {LIT & AM2909  LBL,,}
```

Then in the application program, a statement like the following:

```
    GOSUB   SUBTOTAL;
```

would result in the following semantic actions.

a) first, the CLAUSE's semantic action would be performed: in this case JSRFNO would be inserted into the second field of the AM2909 DEF'ed statement.

b) next, the LITERAL's first semantic action would be performed. The first semantic action, LIT, would cause the label SUBTOTAL (contained in the same location as the LITERAL in the corresponding CLAUSE definition) to be inserted into the active label register.

c) next, the LITERAL's second semantic action would be performed. The active label register (containing SUBTOTAL) would be inserted into the first field of the AM2909 DEF'ed statement.

d) finally, the resulting instruction source statement would be inserted into the output file.

The result would look like this:

```
      AM2909    SUBTOTAL,JSRFNO,;
```

CHAPTER 3

SYNTAX-PHASE SOURCE PROGRAM STATEMENTS

3.1 INTRODUCTION

This chapter contains brief descriptions and formats of all of the types of PSI_S pseudo operations that can be used in PSI_S Syntax-phase source files.

3.2 PRINTING CONTROL STATEMENTS

3.2.1 LIST and NOLIST Statements

A LIST statement is used to turn on listing parameters while a NOLIST statement is used to turn off listing parameters.

The format of a LIST statement is:

```
LIST {p1{,p2{,...,pn}}}
```

and the format of a NOLIST statement is:

```
NOLIST p1,p2,...,pn
```

where p1, p2,...,pn are LIST and NOLIST parameters. The possible listing parameters are shown in table 7-1.

3.2.2 TITLE and TITLE2 Statements

The TITLE and TITLE statements are used to define a user title to be printed on each page of the program listing. A user title and subtitle of up to 50 characters each can be defined by:

```
TITLE title text, title2 text
```

or by

```
TITLE title_text
TITLE2 title2_text
```

where title_text is the title and title2_text is the subtitle.

3.2.3 LINES Statement

LINES is used to define the maximum number of lines to be printed on a page. The form of the LINES statement is:

```
LINES n
```

where n is a decimal number between 8 and the maximum number of lines that can be printed on the form.

3.2.4 EJECT Statement

The EJECT statement causes a page eject to be inserted into the listing. The form of the EJECT statement is:

```
EJECT
```

3.2.5 WIDTH Statement

The WIDTH statement defines the number of columns per line in the program listing.

The format of a WIDTH statement is:

```
WIDTH n
```

where n is a decimal number defining the column width of the listing which can be no larger than 132.

3.3 INCLUDE STATEMENT

The INCLUDE statement defines a secondary file as a source file. When the PSI_S program reaches an INCLUDE statement, it processes the statements contained in the secondary file named in the INCLUDE statement. The form of an INCLUDE statement is:

```
INCLUDE filename
```

where filename is the name of the secondary source file to be included.

Most statements in the secondary source file are processed in the same manner as they would have been had they been listed in the primary file. There are two exceptions to this. When the PSI_S program encounters an END statement in a secondary file or reaches the end of the file, it returns to processing the primary file. Also, the secondary file is not permitted to contain any INCLUDE statements.

3.4 END STATEMENT

The PSI_S program will continue processing statements until it encounters an END statement (or an End Of File marker) in a source file. The form of the END statement is:

```
END
```

intentionally left blank

CHAPTER 4

COMPILE-PHASE SOURCE PROGRAM STATEMENTS

4.1 INTRODUCTION

This chapter contains brief descriptions and formats of all of the types of PSI_C statements, other than comment statements, that can be used in the PSI_C compiler-phase source programs.

4.2 DEFINITON STATEMENTS

4.2.1 EQU Statement

The EQU is used to equate a label name to an expression. The form for the EQU statement is:

```
label: EQU expression
```

where label defines the constant name which is equated to the value of expression.

4.2.2 SET Statement

The SET is used to temporarily equate a label to the value of an expression. The label retains this value until a later SET assigns another value to the same label. The format for the SET statement is:

```
label: SET expression
```

where label is temporarily equated to the value of expression.

4.3 PRINTING CONTROL STATEMENTS

4.3.1 LIST and NOLIST Statements

A LIST statement is used to turn on listing parameters while a NOLIST statement is used to turn off listing parameters.

The form of the LIST and NOLIST statements are:

```
LIST {p1{,p2{,...,pn}}}
NOLIST {p1{,p2{,...,pn}}}
```

where p1, p2,...,pn are LIST and NOLIST parameters. The possible listing parameters are shown in table 8-1.

4.3.2 TITLE and TITLE2 Statements

These statements are used to define a user title to be printed on each page of the program listing. A user title and subtitle of up to 50 characters each can be defined by:

```
TITLE{,x} title text, title2 text
```

or by

```
TITLE{,x} title_text
TITLE2{,x} title2_text
```

where title_text defines the title and title2_text defines the subtitle.

where the optional parameter x can take on one of two values - either "C" or "A". The "C" will cause the compiler-phase to execute the TITLE and TITLE2 statements, and the "A" will cause the TITLE and TITLE2 statements to be copied into the Assembly-phase source file. If the parameter is omitted then the TITLE and TITLE2 statements will be executed in the Compile-phase and copied into the Assembly-phase.

4.3.3 HEAD Statement

HEAD is used to define the headings for object code. Obect code headings are printed above the object code on each page of the listing when the assebler-phase LIST F parameter is on. The form of the HEAD statement is:

```
HEAD "heading_text"
```

where heading_text consists of any column heading including embedded blanks.

4.3.4 FORM Statement

The FORM statement specifies the object code listing format. The form of the FORM statement is:

```
FORM fffffffff......
```

where each f is either one of the digits 1, 3, 4 (defining a binary, octal, or hexidecimal digit) or the letter, B, specifying a blank.

4.3.5 LINES Statement

LINES is used to define the maximum number of lines to be printed on a page. The form of the LINES statement is:

```
LINES{,x} n
```

where n is a decimal number greater than 8 and less than the maximum number of lines that can be printed on the form.

where the optional parameter x can take on one of two values - either "C" or "A". The "C" will cause the compiler-phase to execute the LINES statement, and the "A" will cause the LINES statement to be copied into the Assembly-phase source file. If the parameter is omitted then the LINES statement will be executed in the Compile-phase and copied into the Assembly-phase.

4.3.6 SPACE Statement

A SPACE statement causes the insertion of one or more blank lines into the listing. The form of a SPACE statement is:

```
SPACE{,x} expression
```

where n is a decimal number defining the number of blank lines to be inserted.

where the optional parameter x can take on one of two values - either "C" or "A". The "C" will cause the compiler-phase to execute the SPACE statement, and the "A" will cause the SPACE statement to be copied into the Assembly-phase source file. If the parameter is omitted then the SPACE statement will be executed in the Compile-phase and copied into the Assembly-phase.

4.3.7 EJECT Statement

The EJECT statement causes a page eject to be inserted into the listing. The form of the EJECT statement is:

```
EJECT{,x}
```

where the optional parameter x can take on one of two values - either "C" or "A". The "C" will cause the compiler-phase to execute the EJECT statement, and the "A" will cause the EJECT statement to be copied into the Assembly-phase source file. If the parameter is omitted then the EJECT statement will be executed in the Compile-phase and copied into the Assembly-phase.

4.3.8 WIDTH Statement

The WIDTH statement defines the number of columns per line in the program listing.

The format of a XXX statement is:

```
WIDTH{,x} n
```

where n is a decimal number defining the column width of the listing. This width must be less than 133.

where the optional parameter x can take on one of two values - either "C" or "A". The "C" will cause the compiler-phase to execute the WIDTH statement, and the "A" will cause the WIDTH statement to be copied into the Assembly-phase source file. If the parameter is omitted then the WIDTH statement will be executed in the Compile-phase and copied into the Assembly-phase.

4.3.9 TAB Statement

The TAB statement defines the number of blank characters, n, to be substitued for each TAB character found in a MACRO body statement. The form for the TAB statement is:

```
TAB n
```

where n is a decimal number between 1 and 10.

4.4 CONDITIONAL COMPILATIION STATEMENTS

4.4.1 General

Conditional compilation statements are used to compile a group of statements only if some specified condition is satisfied or to compile one group of statements if a specified condition is satisfied and an alternate group of statements if that condition is not satisfied.

The general format for an IF-type..ELSE..endif sequence is:

```
IF-type statement
statement
    .
    .
statement
ELSE
statement
    .
    .
statement
ENDIF
```

Possible IF-type statements include IF, IFC, IFNC, IFD, and IFND statements.

4.4.2 IF Statement

The form of the IF statement is:

```
IF expression
```

where expression is evaluated to determine whether it is non-zero. The IF condition is satisfied, by definition, if any non-zero value is obtained. Otherwise, the IF condition is not satisfied.

4.4.3 IFC and IFNC Statements

The format of the IFC statement is:

```
IFC string1, string2 {,start,end}
```

where string1 and string2 are two strings of characters, and start and end are optional pointers to character positions in string1 and string2

A character by character comparison of string1 with string2 is performed. If each character of string1 that is compared with the corresponding character of string2 is identical to that character, then by definition, an IFC statement is satisfied . An IFNC statement is if at least one of the characters of string1 that is compared with a string2 character differs from that string2 character.

4.4.4 IFD and IFND Statements

The format of the IFD statement is:

IFD symbol

The statement is satisfied if symbol has been defined by a prior statement in the assembly phase program or by a definition file statement.

The format of the IFND statement is:

IFND symbol

The statement is satisfied if symbol has not been defined by a prior statement in the assembly phase program or by a definition file statement.

4.5 MACRO DEFINITION AND CALL STATEMENTS

4.5.1 MACRO Definitions

A MACRO definiton opens with a MACRO statement and closes with an ENDM statement. Optionally, the MACRO statement may be followed by one or more LOCAL statements. Statements between the LOCAL statements and the ENDM statement or, if no LOCAL statements are required, between the MACRO statement and the ENDM statement, constitute the bogy of the MACRO.

4.5.2 MACRO Definition Statement Formats

The format of a MACRO statement is:

symbol: MACRO {f1{,f2,...,fn}}

where symbol is the MACRO name

and f1, f2, ...,fn are optional parameters.

The format for the LOCAL statement is:

LOCAL symbol1{,symbol2,...,symboln}

where symbol1, symbol2, ...,symboln are symbolic address labels used in the body of the MACRO.

The format of the ENDM statement is:

```
{label:} ENDM
```

where label is an optional symbolic address label which is assigned the assembly program counter address pointing to the location following the last word of the expanded MACRO.

The format of the EXITM statement is:

```
{label:} EXITM
```

where label is as defined for the ENDM statement.

4.5.3 MACRO Call Statements

The format of a MACRO call is:

```
{symbol:} MACRO name {s1},{s2},...,{sn}
```

where symbol is an optional symbolic address label

and s1, s2,...,sn are individually optional actyal parameters substituted respectively for formal parameters f1, f2, ...,fn in the MACRO definition.

4.6 INCLUDE STATEMENT

An INCLUDE statement names a secondary source file. When the definition or assembly phase program reaches an INCLUDE statement, it processes the statements contained in the file named in the INCLUDE statement. In general, the statements in the secondary source file are processed in exactly the same manner as they would have been had they been listed in the primary file starting at the point where the INCLUDE statement is listed.

There are two exceptions to this:

1. When the program reads the END statement in the secondary source file or reaches the end of the file, it continues with the processing of the primary source file starting with the statement following the INCLUDE statement.

2. The secondary source file is not permitted to contain any INCLUDE statements.

the format of an INCLUDE statement is:

```
INCLUDE{,x} filename
```

where filename is the name of the secondary source file to be included.

where the optional parameter x can take on one of two values - either "C" or "A". The "C" will cause the compiler-phase to execute the INCLUDE statement, and the "A" will cause the INCLUDE statement to be copied into the Assembly-phase source file. If the parameter is omitted then the INCLUDE statement will be executed in the Compile-phase and copied into the Assembly-phase.

4.7 PROGRAM COUNTER CONTROL STATEMENTS

4.7.1 ORG Statement

An ORG statement sets the assembly program counter to a specified value. The value specified in an ORG statement must be larger than the current value of the assembly program counter. The format of the ORG statement is:

```
{symbol:} ORG expression
```

where

symbol is an optional symbolic address label

expression is an expression which is evaluated to determine the setting of the assembly program counter.

4.7.2 RES Statement

A RES statement causes a specified number to be added to the current assembly program counter so as to reserve a block of memory locations. The format of the RES statement is:

```
{symbol:} RES expression
```

where

symbol is an optional symbolic address label which is assigned to the first location of the reserved block of addresses and expression is evaluated to determine the number of locations to be reserved.

4.7.3 ALIGN Statement

An ALIGN statement causes the assembly program counter to be advanced to the next value located on a specified boundary. The boundary is specified in terms of a factor of which the boundary is an integral multiple. The format of the ALIGN statement is:

```
{symbol:} ALIGN expression
```

where

symbol is an optional symbolic address label to be assigned to the aligned assembly program counter value

and expression is evaluated to determine the boundary factor.

4.8 INSTRUCTION/DATA SOURCE STATEMENTS

4.8.1 FF Statements

FF statements are used to generate executable microinstructions without reference to instruction format or instruction sub-format names. This requires specification of a constant or don't care values for every field of the instruction. The format of the FF statement is:

```
{symbol:} FF field1{,field2}....{,
/fieldn}
```

4.8.2 DATA Statements

The DATA statement provides the means of setting up a data constant (literal) in a separate microword to be accessed under control of the executable instruction in a preceding microword. The format of the DATA statement is:

```
{symbol:} DATA expression
```

where

symbol is an optional symbolic address label

and expresion is an expression which is evaluated to determine the value to be placed in the microword.

4.8.3 DUP Statements

The DUP statement is used to generate multiple lines of instructions or data. The DUP statement causes the instruction or data word on the line following the dup statement to be repeated a specified number of times. The format of the DUP statement is:

```
DUP expression
```

where

expression is evaluated to determine the number of duplications.

4.9 RELOCATION DEFINITION STATEMENTS

4.9.1 .REL Statement

In a relocatable assembly-phase source program, the .REL statement must be entered prior to the first statement which affects the assembly program counter. Thus, the .REL statement must be entered before any machine instruction source statement or any ORG statement. The format of the .REL statement is:

```
.REL
```

4.9.2 EXTRN Statement

An EXTRN statement is used to declare one or more symbols as extrenal symbols. An extrenal symbol is a symbol not defined in the definition file or in the current assembly phase source program but in another program file that is subsequently to be linked to this one. Any given symbol can only be listed in one EXTRN statement of a particular file. A symbol that is declared as EXTRN in one or more files must be listed in a PUBLIC statement in the file in which it is defined. The format of an EXTRN statement is:

```
EXTRN symbol1{,symbol2{,...,symboln}}
```

4.9.3 PUBLIC Statement

A PUBLIC statement makes the listed symbols available for declaration as EXTRN symbols in other assembly-phase source program files. To be declared PUBLIC, a symbol must be defined in the current assembly-phase source program file or in the definiton file. A symbol can only be declared PUBLIC in one PUBLIC statement in any group of files that are to be linked. The format of a PUBLIC statement is:

```
PUBLIC symbol1{,symbol2{,...,symboln}}
```

4.10 MAP STATEMENT

The MAP statement is appropriate only in an absolute assembly-phase source program and is ignored if entered in a relocatable assembly-phase source program.

When a MAP statement is entered in an absolute assembly-phase source program, a separate entry point object file is created. The name, srcfile.map, is automatically assigned to this file, where srcfile is the name of the assembly-phase source program file. A listing of the entry point map is provided at the end of the list file.

The entry point map object file consists of the addresses assigned to all entry point symbolic address labels. A symbolic address label is specified as an entry point by terminating the symbol field of the source statement with a double colon (::). The format of the MAP statement is:

```
MAP address,width
```

where address (in decimal) is the base address of the entry point map

and width (in decimal) is the width of the entry point map object words.

4.11 END STATEMENT

When the compile-phase program encounters an END statement in a source file, it interprets this as the end of the file. If no END statement is encountered, the program will continue processing statements until the end of the source file is encountered. The format of the END statement is:

```
END
```

4.12 DCARE STATEMENT

The DCARE statement is used in the assembly-phase source program to assign values to don't care bits.

CHAPTER 5

EXECUTION OF SYNTAX-PHASE PROGRAM

5.1 STARTING THE SYNTAX-PHASE PROGRAM

In order to start the execution of the syntax-phase program, the following information must be specified:

the syntax-phase program name (psi_s)

the syntax-phase source file name (synfile)

the definition-phase file name (srcfile)

the syntax table file name (tabfile)

the list file name (listfile) or (null) to inhibit the generation of a list file

The operator can optionally specify the states of one or more LIST parameters. These specifications override the default states of these parameters but are, in turn, overriden by any contrary LIST or NOLIST statements in the source file.

The user also has the option of changing the semantic actions to support cascadable processors. The width of the microword (in processors) is by default set to one. Any other value can be specified by an integer preceeded by a pound sign (#).

If the selected syntax table size is not large enough to support the actual size of the syntax tables defined in the source program then a Table Overflow error will occur during the running of the program. If extended memory is available, then the syntax tables can be moved to extended memory. This option is selected by an X (see table 5-1). Note: extended memory must be based at H100000.

The user can start the execution of the syntax-phase program by entering all of the required and optional information in a single statement having the following format:

```
psi_s synfile srcfile tabfile lstfile {-list parameter
settings}{#cascadable processors}<CR>
```

Example 1:

```
psi_s Modl W24ns Modl Modl.lst -EX #2<CR>
```

NOTES:

1. If the synfile, srcfile, and tabfile entered by the user don't include extensions, the program appends .syn to synfile, .src to srcfile, and .tab to tabfile. If the user does include extensions, these are accepted by the program as entered. The program accepts lstfile, as entered, with or without an extension. Thus, in the example above, the program uses syntax-phase source file Modl.syn and definition-phase source file W24ns.src and names the syntax table file Modl.tab and the list file Modl.lst.

2. A dash (-) must precede the first LIST parameter whose state is specified. Each LIST parameter that is named is turned off if preceded by a caret(^) or is turned on if not preceded by a caret. For example, -^ES turns off LIST parameter E and turns on LIST parameter S. Refer to table 5-1 for LIST/NOLIST parameters with default values as applicable to the compile phase.

As an alternative to supplying all of the required and optional information, the user can invoke the program with a statement of the form:

```
psi_s {synfile}{-LIST parameter settings}
{#cascadable processors}<CR>
```

where the only mandatory components of the statement are psi_s and <CR>.

The program responds by displaying a title/copyright message and a sequence of prompts for the file names not included in the opening statement. If no LIST parameter settings are included in the opening statement, the program prompts for such settings. If any LIST parameter settings are included in the opening statement, the program does not prompt for additional LIST parameter settings. The program does not prompt for a symbol table size. Thus, to select a size other than the default value of ???, the symbol table size selection must be specified in the opening statement.

If the syntax-phase source file name is not specified in the opening statement, the first prompt is as follows:

```
Enter source file name [.syn]:
```

where [.syn] indicates the extension that the program will automatically append to the specified file name if no other extension is specified by the user.

If the user keys <CR> without entering a file name in response to the syntax-phase source file name prompt, the program terminates.

The remaining file name prompts display default values within square brackets. These default names are generated by the program by appending an appropriate extension (.src, .tab, .lst) to the syntax-phase source file name specified by the user.

The definition-phase source file name prompt is as follows:

```
Enter definition-phase source file name [srcfile.src]:
```

The user now enters his selected definition-phase source file name followed by <CR> or enters <CR> only to accept the default name displayed in square brackets.

The syntax table file name prompt is as follows:

```
Enter syntax table file name [tabfile.tab]:
```

The user now enters his selected syntax table file name followed by <CR> or enters <CR> only to accept the default name displayed in square brackets.

The list file name prompt is as follows:

```
Enter list file name [lstfile.lst]:
```

The user now enters his selected list file name followed by <CR> or enters <CR> only to accept the default name displayed in square brackets.

The LIST selection prompt is as follows:

```
Enter LIST options:
```

The user now enters any desired LIST settings exactly as in the opening statement except that the dash (-) preceding the first setting is omitted and that <CR> is keyed to complete the entry. If no LIST option settings are desired, the user keys <CR> in response to the prompt.

```
Example 2:     psi_s   Modl   #2<CR>
```

NOTE: This entry sequence produces the same results as the single statement of Example 1, above. Program prompts are in bold type. x.x = program revision level.

```
PENGUIN SOFTWARE Microcode compiler: Syntax Phase v. x.x
Copyright PENGUIN SOFTWARE,Inc. 1988.

Enter definition-phase source file name [Modl.src]: W24ns
<CR>

Enter syntax table file name [Modl.tab]:  <CR>

Enter list file name [Modl.lst]:  <CR>

Enter LIST options:E <CR>
```

NOTE: When the user types in a definition-phase source file name or a syntax table file name other than that in square brackets and does not type in an extension, the program appends the extension displayed in the square brackets, Thus, in the above example, the definition-phase source file name becomes W24ns.src.

5.2 ERROR MESSAGES

If an error is encountered during the execution of the syntax-phase program, an error message is generated and inserted in the list file. If the LIST E parameter is set, error messages are also listed on the system console as they are generated. Upon completion of the program, the error total is reported on the system console and is also placed at the end of the list file.

Error messages that can be generated during the execution of the syntax-phase program are listed in table 5-2.

Table 5-1. Syntax-Phase LIST/NOLIST Parameters

Parameter	Description	Default State
S	List text (when this is off no source statements are listed	On
E	List source lines with errors at user's console as well as in output listing	Off
X	Use extended memory for syntax tables	Off

Table 5-2. Syntax-Phase error messages

error message	Description
ERR001	There are too many CLAUSE defintions.
ERR002	This CLAUSE contains an undefined LITERAL(?).
ERR003	This statement has an illegal syntax.
ERR004	This statement is missing an "{".
ERR005	This statement is missing an "(".
ERR006	This statement requires a non-negative number.
ERR007	This staement is missing an ")".
ERR008	Empty CLAUSEs are illegal.
ERR009	"string" can't have an empty string.
ERR010	This CLAUSE contains a string that is not qouted and not defined ARRAY.
ERR011	The number of fields in a statement doesn't match the DEF statement.
ERR012	There are too many ARRAY labels.
ERR013	There are too many ARRAY definitions.
ERR014	The syntax of this statement requires a string at this location.
ERR015	This ARRAY names appears to have more than one definition.
ERR016	The syntax of this statement requires a "=" in this location.

Table 5-2. Syntax-Phase error messages(cont.)

error message	Description
ERR017	This ARRAY definition has too many columns.
ERR018	The number of columns in an ARRAY must match the ARRAY definition.
ERR019	This statement is missing an "}".
ERR020	This program contains nested includes
ERR021	This program contains too many LITERAL definitions.
ERR022	This LITERAL is definitoned more than once.
ERR023	This program has too many symbols.
ERR024	This ARRAY is dimensioned and should be parameterized when referrenced.
ERR025	This program contains nested IF-type statements.
ERR026	This semantics definition should contain semantic actions.
ERR027	There are several legitimate tokens that could appear here but this isn't one of [illegible].
ERR028	IF-THEN-ELSE-ENDIF statements requir an IF preceeding ELSE.
ERR029	IF-THEN-ELSE-ENDIF statements require an IF preceeding ENDIF.
ERR030	The should have a PSEUDO OPCODE at this location.
ERR031	This value exceeds the maximum width.
SYS00n	Compiler Error

CHAPTER 6

EXECUTION OF COMPILE-PHASE PROGRAM

6.1 STARTING THE COMPILE-PHASE PROGRAM

In order to start the execution of the compile-phase program, the following information must be specified:

the compile-phase program name (psi_c)

the compile-phase source file name (cmpfile)

the syntax definition file name (tabfile)

the assembly source output file name (srcfile)

the list file name (listfile) or (null) to inhibit the generation of a list file

The operator can optionally specify the states of one or more LIST parameters. These specifications override the default states of these parameters but are, in turn, overriden by any contrary LIST or NOLIST statements in the source file.

The user can start the execution of the compile-phase program by entering all of the required and optional information in a single statement having the following format:

```
psi_c cmpfile tabfile srcfile lstfile {-list parameter
settings}<CR>
```

Example 1:

```
psi_c Modl W24ns Modl Modl.lst -C <CR>
```

NOTES:

1. If the cmpfile, tabfile, and srcfile entered by the user don't include extensions, the program appends .cmp to cmpfile, .tab to tabfile, and .src to srcfile. If the user does include extensions, these are accepted by the program as entered. The program accepts lstfile, as entered, with or without an extension. Thus, in the example above, the program uses compile-phase source file Modl.cmp and syntax table file W24ns.tab and names the assembly source output file Modl.src and the list file Modl.lst.

2. A dash (-) must precede the first LIST parameter whose state is specified. Each LIST parameter that is named is turned off if preceded by a caret(^) or is turned on if not preceded by a caret. For example, -^YC turns off LIST parameter Y and turns on LIST parameter C. Refer to table 6-1 for LIST/NOLIST parameters with default values as applicable to the compile phase.

As an alternative to supplying all of the required and optional information, the user can invoke the program with a statement of the form:

```
psi_c{cmpfile}{-LIST parameter settings}<CR>
```

where the only mandatory components of the statement are psi_c and <CR>.

The program responds by displaying a title/copyright message and a sequence of prompts for the file names not included in the opening statement. If no LIST parameter settings are included in the opening statement, the program prompts for such settings. If any LIST parameter settings are included in the opening statement, the program does not prompt for additional LIST parameter settings.

If the compile-phase source file name is not specified in the opening statement, the first prompt is as follows:

```
Enter source file name [.cmp]:
```

where [.cmp] indicates the extension that the program will automatically append to the specified file name if no other extension is specified by the user.

If the user keys <CR> without entering a file name in response to the compile-phase source file name prompt, the program terminates.

The remaining file name prompts display default values within square brackets. These default names are generated by the program by appending an appropriate extension (.tab, .src, .lst) to the compile-phase source file name specified by the user.

The syntax table file name prompt is as follows:

Enter syntax table file name [tabfile.tab]:

The user now enters his selected syntax table file name followed by <CR> or enters <CR> only to accept the default name displayed in square brackets.

The assembly source output file name prompt is as follows:

Enter assembly source output file name [srcfile.src]:

The user now enters his selected assembly source output file name followed by <CR> or enters <CR> only to accept the default name displayed in square brackets.

The list file name prompt is as follows:

Enter list file name [lstfile.lst]:

The user now enters his selected list file name followed by <CR> or enters <CR> only to accept the default name displayed in square brackets.

The LIST selection prompt is as follows:

Enter LIST options:

The user now enters any desired LIST settings exactly as in the opening statement except that the dash (-) preceding the first setting is omitted and that <CR> is keyed to complete the entry. If no LIST option settings are desired, the user keys <CR> in response to the prompt.

Example 2:   psi_c   Modl   <CR>

NOTE: This entry sequence produces the same results as the single statement of Example 1, above. Program prompts are in bold type. x.x = program revision level.

```
PENGUIN SOFTWARE Microcode compiler: Compile Phase v. x.x
Copyright PENGUIN SOFTWARE,Inc. 1988.

Enter syntax table file name [Modl.tab]: W24ns <CR>

Enter assembly source output file name [Modl.src]:  <CR>
Enter list file name [Modl.lst]:  <CR>

Enter LIST options:C <CR>
```

NOTE: When the user types in a syntax table file name or an assembly source output file name other than that in square brackets and does not type in an extension, the program appends the extension displayed in the square brackets, Thus, in the above example, the syntax table file name becomes W24ns.tab.

6.2 ERROR MESSAGES

If an error is encountered during the execution of the compile-phase program, an error message is generated and inserted in the list file. If the LIST E parameter is set, error messages are also listed on the system console as they are generated. Upon completion of the program, the error total is reported on the system console and is also placed at the end of the list file.

Error messages that can be generated during the execution of the compile-phase program are listed in table 6-2.

Table 6-1. Compile-Phase LIST/NOLIST Parameters

Parameter	Description	Default State
L	List text (when this is off no source or assembly source output elements are listed	On
C	List source lines with errors at user's console as well as in output listing	Off
P	Wraparound to next line when line exceeds maximum width. (If off, lines that exceed maximum width are truncated.)	Off
J	List assembly sourceoutput immediately following compiler-phase source if L is on*	Off

Table 6-1. Compile-Phase error messages

error message	Description
ERR001	This sentance contains and empty clause.
ERR002	The syntax of a sentence requires either a number, a string, an "&", or a ";".
ERR003	The field defintiions in the semantic actions of this sentence are in conflict.
ERR004	The semantic actions of this sentence contain too many fields.
ERR005	Too many items were pushed onto the stack.
ERR006	Too many items were poped off the stack.
ERR007	This sentence contains a clause which was not defined in the syntax phase.
ERR008	This sentence contains a clause which was defined more than once in the syntax phase

Table 6-1. Compile-Phase error messages(cont.)

error message	Description
ERR009	This program contains too many symbols.
ERR010	This PSEUDO OPCODE has an illegal parameter.
ERR011	This PSEUDO OPCODE requires a number parameter.
ERR012	This number is too large for this PSEUDO OPCODE.
ERR013	This program contain nested INCLUDE's.
ERR014	This LIST OPCODE contains an unknown parameter.
ERR015	This statement contains an unrecognized string of characters.
ERR016	This program contains too many symbols.
ERR017	This sentence has too many labels (some of which may be automatically generated by semantic actions.
SYS00n	These messages indicate an error in the compiler has been detected.

APPENDIX A

EXAMPLE APPLICATION PROGRAM

```
;
;       FILENAME        CAPSHW.CMP
;
; THIS IS A TEST PROGRAM FOR THE MICRO COMPILER
; BASED ON THE AMD PROGRAM
;
            ORG  0
;
; INITIALIZE DATA REGISTERS
;
            R0 = 15;
            R1 = 9;
            R2 = 0;
;
; INITIALIZE BIT COUNTER AND TOTAL
;
            CNTR = 4;
            TOTAL = 0;
;
; COUNT ONES
;
            REPEAT;
                IF(R0(SRA) IS ODD) CALL UPTOTAL;
                IF(R1(SRA) IS ODD) CALL UPTOTAL;
                IF(R2(SRA) IS ODD) CALL UPTOTAL;
            UNTIL(DEC(CNTR) = ZERO);
;
; LOOP WHILE OUTPUTTING TOTAL
;
            REPEAT;
                OUTPUT TOTAL&
            UNTIL(FOREVER);
;
; ROUTINE INCREMENTS ONES COUNTER
;
UPTOTAL:    ROUTINE&
                INC(TOTAL)&
            RETURN;
            END
```

intentionally left blank

APPENDIX B

EXAMPLE SYNTAX DEFINITION PROGRAM

```
TITLE This is a test file
TITLE2  for the syntax generator
ARRAY IFCON(1,2) = {
        IFNOT,ZERO{AM2909  ,BRFNO,};
        IFNOT,OVER{AM2909  ,BROVR,};
        IF,ZERO{AM2909  ,BRFEQ0,};
        IF,F3{AM2909  ,BRF3,};
        IF,C4{AM2909  ,BRCOUT,};
}
ARRAY MCS1 = {
        CONT{AM2909  ,CONT,};
        RETURN{AM2909  ,RTS,};
        PUSH{AM2909  ,PUSH,};
        POP{AM2909  ,POP,};
}
ARRAY MCS2 = {
        ZERO{AM2909  ,LOOPFNO,};
        C4{AM2909  ,LOOPCOUT,};
}
ARRAY ABREGS(1) = {
        R0{AM2901  ,,,,,,R0,R0,};
        R1{AM2901  ,,,,,,R1,R1,};
        R2{AM2901  ,,,,,,R2,R2,};
        TOTAL{AM2901  ,,,,,,R3,R3,};
        CNTR{AM2901  ,,,,,,R4,R4,};
        R5{AM2901  ,,,,,,R5,R5,};
        R6{AM2901  ,,,,,,R6,R6,};
        R7{AM2901  ,,,,,,R7,R7,};
        R8{AM2901  ,,,,,,R8,R8,};
        R9{AM2901  ,,,,,,R9,R9,};
        R10{AM2901  ,,,,,,R10,R10,};
        R11{AM2901  ,,,,,,R11,R11,};
        R12{AM2901  ,,,,,,R12,R12,};
        R13{AM2901  ,,,,,,R13,R13,};
        R14{AM2901  ,,,,,,R14,R14,};
        R15{AM2901  ,,,,,,R15,R15,};
}
ARRAY BREGS(1) = {
        R0{AM2901  ,,,,,,,R0,};
        R1{AM2901  ,,,,,,,R1,};
        R2{AM2901  ,,,,,,,R2,};
        TOTAL{AM2901  ,,,,,,,R3,};
        CNTR{AM2901  ,,,,,,,R4,};
```

```
                R5{AM2901  ,,,,,,,R5,};
                R6{AM2901  ,,,,,,,R6,};
                R7{AM2901  ,,,,,,,R7,};
                R8{AM2901  ,,,,,,,R8,};
                R9{AM2901  ,,,,,,,R9,};
                R10{AM2901  ,,,,,,,R10,};
                R11{AM2901  ,,,,,,,R11,};
                R12{AM2901  ,,,,,,,R12,};
                R13{AM2901  ,,,,,,,R13,};
                R14{AM2901  ,,,,,,,R14,};
                R15{AM2901  ,,,,,,,R15,};
        }
        LITERAL(1) = {LIT & AM2909  LBL,,};
        LITERAL(2) = {LIT & DIN  ,LBL};
        LITERAL(3) = {};
                          {LIT & AM2909  LBL,,};
CLAUSE {ABREGS(1) "AND" LITERAL(2)}
            {AM2901  ,,,DA,,AND,,,}
CLAUSE {BREGS(1) "OR" LITERAL(2)}
            {AM2901  ,,,ZB,,OR,,,}
CLAUSE {BREGS(1) "=" BREGS(1) "+" "ONE"}
            {AM2901  ,RAMF,,ZB,CN1,ADD,,,}
CLAUSE {"INC(" BREGS(1) ")"}
            {AM2901  ,RAMF,,ZB,CN1,ADD,,,}
CLAUSE {IFCON(1) IFCON(2) "GOTO" LITERAL(1)}
CLAUSE { MCS1 }
CLAUSE {"GOSUB" LITERAL(1)}{AM2909  ,JSR,}
CLAUSE {"CALL" LITERAL(1)}{AM2909  ,JSR,}
CLAUSE {"GOTO" LITERAL(1)}{AM2909  ,BR,}
CLAUSE {"GOTO" "SWITCH"}{AM2909  ,BM,}
CLAUSE {"GOTO" "FILE"}{AM2909  ,STKREF,}
CLAUSE {"IFNOT" "ZERO" "GOSUB" LITERAL(1)}
            {AM2909  ,JSRFNO,}
CLAUSE {"IF" MCS2 "END" "LOOP" "AND" "POP"}
CLAUSE {BREGS(1) "=" BREGS(1) "-" "ONE"}
            {AM2901  ,RAMF,,ZB,CNO,SUBR,,,}
CLAUSE {"DEC(" BREGS(1) ")"}
            {AM2901  ,RAMF,,ZB,CNO,SUBR,,,}
CLAUSE {BREGS(1) "=" "SRA" BREGS(1)}
            {AM2901  ,RAMD,,ZB,,OR,,,}
CLAUSE {"SRA(" BREGS(1) ")"}
            {AM2901  ,RAMD,,ZB,,OR,,,}
CLAUSE {"REPEAT"}{MAK & PSH & PLB}
CLAUSE {"UNTIL(FOREVER)"}{PPS & AM2909  LIT,BR,}
CLAUSE {"ROUTINE"}
CLAUSE {"OUTPUT" BREGS(1)}
            {AM2901  ,,,ZB,,OR,,,&DIN ,0}
```

```
CLAUSE {"UNTIL(DEC("BREGS(1)")=ZERO)"}
             {AM2901  ,RAMF,,ZB,CNO,SUBR,,,};
             {PPS & AM2909  LIT,BRFNO,};
CLAUSE {"IF(" BREGS(1) "(SRA) IS ODD)GOSUB" LITERAL(3)}
             {AM2901  ,,,DA,,AND,,,&DIN  ,1}
             {AM2909  ,JSRFNO,&AM2901  ,RAMD,,ZB,,OR,,,}
CLAUSE {"IF(" BREGS(1) "(SRA) IS ODD)CALL" LITERAL(3)}
             {AM2901  ,,,DA,,AND,,,&DIN  ,1}
             {AM2909  ,JSRFNO, & AM2901  ,RAMD,,ZB,,OR,,,}
CLAUSE {BREGS(1) "=" LITERAL(2)}{AM2901  ,RAMF,,DZ,,OR,,,}
```

intentionally left blank